A Three Level Framework for Adapting Component-Based Systems

Nicolas Pessemier[1], Olivier Barais[1], Lionel Seinturier[1], Thierry Coupaye[2], and Laurence Duchien[1]

[1] INRIA Futurs, USTL-LIFL,Jacquard, Villeneuve d'Ascq, France
{pessemie, seinturi, barais, duchien}@lifl.fr
[2] France Telecom R&D, France
thierry.coupaye@rd.francetelecom.com

Abstract. This paper deals with the issue of software adaptation. We focus on Component-Based Software Development including Architecture Description Languages, and clearly identify three levels of adaptation. We argue that capturing functional and non-functional changes in a system requires various types of adaptation tools working at different granularities and times in the system lifecycle, with various actors.

1 Introduction

Adaptation has always been an important challenge for software engineering. Systems have to be continuously revised to address new functional or non-functional requirements, changing environment. The need for adaptation may appear at any time in the software lifecycle: development, deployment, supervision and maintenance (evolutive, corrective). Changes that can be anticipated at development and deployment time are referred to as static adaptation, and changes applied at execution time without stopping the system, as dynamic adaptation. Maintenance changes can be done statically or dynamically.

Researchers have come with various solutions to address the issue of adaptation. These include model transformations in Model Driven Engineering [1], approaches based on reflection [2], adaptive agent platforms [3], object and component-based approaches that propose open containers in EJB and CCM models [4], component assembly reconfiguration [5, 6], adaptors for component [7], and Aspect Oriented Programming techniques (AOP) [8, 9].

In this position paper, we propose a multi-level model that aims at capturing static and dynamic changes in a Component-Based Software Development (CBSD) and Architecture Description Languages (ADLs) context. We promote the use of AOP techniques to enable the integration for both functional and non-functional adaptation. Context-aware and auto-adaptive mechanisms are out of the scope of this paper.

The paper is structured as follows. Section 2 identifies the different adaptation levels, and describes the adaptation techniques currently available at each level. Section 3 shows how our recent work integrating components and aspects, Fractal

Aspect Component (FAC) [10] addresses one of the levels identified in Section 2. Section 4 explains how we envision a framework that captures all of these levels. Finally, Section 5 gives conclusions and future work.

2 Three levels of adaptation

Numerous component models have emerged the last two decades, targeting applications, systems, middleware and operating systems. In this paper, we focus on Component-Based Software Development (CBSD), which is concerned with the assembly of highly reusable and configurable software components [11], including Architecture Description Languages (ADLs) [12], which clarify and ease the description of component assemblies.

This section identifies the entities that need to be adapted because of changes in functional and non-functional requirements. In the CBSD and ADLs contexts, one may want to adapt a component access points, the bindings between heterogeneous components, the composition of a composite component, or some programs that represent component behavior. We identify three separate levels of adaptation, each of them working at various granularities: architecture, component and program levels of adaptation.

Architecture level adaptation A system architecture defines a plan that clearly represents the system structure, indicating how components are bound together, as well as the nesting relationship between components. Architecture adaptation relies on reconfiguration and recomposition of the component assembly: adding or replacing a component, inserting connectors between heterogeneous components, changing the component hierarchy, and so on.

By considering a software architecture description as a model, model transformation approaches based on Model Driven Engineering (MDE) adapt the architecture through successive transformations. For example, TranSAT is a framework for adapting a software architecture to new concerns through transformations [13].

Transformation is not the only adaptation mechanism. To deal with the assembling of heterogeneous COTS components, connectors that adapt the incoming and outcoming component operations are generally used. For example, Unicon proposes specific adaptors that are connectors, which mediate interactions among components by specifying protocols [14].

Component level adaptation A component is a unit for the management of configuration, security, faults, etc., i.e., a functional entity together with a set of associated non-functional properties. These non-functional properties are typically handled by so-called "containers" in component models such as Enterprise JavaBeans or the CORBA Component Model, or "membranes" in the Fractal component model which embody interception-based mechanisms such as reflection or AOP.

Various techniques based on the interception of the original component behavior are employed to adapt components, such as K-Component [5], the open-container approach [4]. The K-Component model relies on a specific adaptation language (Adaptation Contract Description Language) and an adaptation manager that is aware of any changes and can adapt the base system through structural reflection. The open-container approach enables new technical services to be added to EJB and CCM containers.

Program level adaptation At this level, we consider programs as entities encapsulated by components. Numerous techniques exist to perform program adaptation, such as AOP [8, 9], reflection [2], program transformation [15], e.g. Java byte-code transformations (e.g. ASM [16]).

3 Dynamic component adaptation with FAC

This section presents our solution to the dynamic component adaptation issue with our most recent work, Fractal Aspect Component (FAC), which introduces AOP concepts into the Fractal component model [6]. A previous paper [10] presented the basic elements of the first version of FAC. This Section sums up its features and discusses the issue of software adaptation with FAC. Our previous work on JAC [9] and TranSAT [13] covers the program and architecture levels, respectively.

The first subsection introduces the Fractal component model, then its extension FAC for AOP. Finally, we present how FAC addresses adaptation at the component level described in Section 2.

3.1 Fractal

Fractal [6] is a general software component model with the following mean features:

- dynamic components, interfaces and bindings: components are runtime entities that do exist at runtime and can be manipulated as such for management purpose. Components communicate through bindings between interfaces, which are the only access points to components. A binding is a communication channel between a client interface and a server interface.
- hierarchical components: composite components contain recursively primitive components at arbitrary levels to provide a uniform view of software systems at various levels of abstraction.
- shared components: a (sub)component can be contained in several (super) components,. This is very useful typically to model resources which are intrinsically shared.
- reflexive components: architectural introspection for system monitoring and intercession for dynamic reconfiguration.

– openness: the model is defined as a set of concepts (component, interface, binding, membrane, controller, etc.) embodied in an API. It typically proposes some APIs to configure components assemblies by means of binding (between client and server interfaces), containment and lifecyle (start, stop). Controllers that are optional, can be specialized and extended at will, and of course new controllers can be defined.

Interestingly here, a Fractal component is composed out of a membrane and a content. The content is either a primitive component in an underlying programming language or a set of components. The membrane embodies most of the reflexive capabilities by means of controllers that can export or not control interfaces. In Julia, a Java execution support for Fractal components, a mixin-based mechanism is used to build controllers that are composed, if needed, with interceptors to build membranes that control the encapsulated components. In AOKell, another Java execution support for Fractal component under development by the authors, (AspectJ) aspects are used to program membranes.

3.2 FAC

FAC is an extension of Fractal, which integrates the notion of aspects into the Fractal model. It aims at capturing the crosscutting properties of a system. In FAC, aspects are Fractal components, called Aspect Components, with a specific server interface implementing the AOP Alliance API[3].

Aspect Components are woven and unwoven at run-time. The process of weaving is very similar to the process of binding a functional client interface and a server interface in Fractal. We call a *crosscutting binding* the interaction between a set of components and an aspect component. Crosscutting bindings are defined by an API or at the ADL level. Pointcuts are defined through regular expressions when a crosscutting binding is defined. A pointcut selects the components, interfaces and methods on which the aspect component will apply. FAC allows structural and behavioral pointcuts as we will see in the following subsection.

3.3 Runtime adaptation with FAC

In FAC adding a new behavior consists of writing the new behavior in an *aspect component* and defining where this new behavior applies. The way the new behavior will be triggered can be expressed with structural or behavioral specifications.

Structural elements, such as a method signature, a functional interface, or a component, can be used as joinpoints in the system. Each required or provided operation defined by a component can potentially be intercepted and augmented with new features.

[3] AOP Alliance *http://sourceforge.net/projects/aopalliance* is an open-source initiative to define a common API for AOP frameworks. The API is implemented by Spring and JAC [9].

FAC allows an aspect to be triggered on a specific sequence of external component interactions. Each component interaction is captured by the aspect component that will be triggered if the sequence of interactions matches. Cflows in JAsCo [8], EAOP [17] and AspectJ can similary trigger aspects on protocols. These approaches work at the program level, whereas FAC works at the component level.

4 Towards an integration of the three levels

Our objective is to build a three level model (architecture, component, program), which captures any functional and non-functional changes at any time in the software lifecycle. In our vision, the model needs to address the following issues:

- who realizes the modifications: the architect, the programmer, the administrator,
- when are the modifications applied: static or dynamic adaptation,

Different actors are involved in a system lifecycle. Each actor needs to perform adaptation at the level he works with. For example, an architect would perform architecture and component level adaptation; a programmer would perform program adaptation. The important point here is that each actor needs a way to adapt the system at a step of the lifecycle. The three levels can fulfill these needs. The issue of who will adapt the system when changes need to be performed remains open.

The three levels need to capture both static and dynamic adaptation. Predictable changes can be defined through adaptation policies. If static adaptation is important, unpredictable changes might appear during run-time. Currently runtime changes are addressed by FAC at the component level.

Our objective is to extend FAC to the architecture and the program levels. Previous works around TranSAT and JAC will inspire the extension.

5 Conclusions & future work

We have proposed a three-level model for adaptation in a component-based context. We have shown that in order to capture any changes in a component system, different granularities have to be considered.

The architecture is likely to evolve through transformations and reconfigurations of components interactions and composition. Component interfaces frequently need to be adapted to new requirements. Finally, when changes apply to specific part of a program encapsluted into a component, only this part needs to be updated or intercepted.

Our proposal uses AOP techniques at the three levels due to its ability to ease the integration of crosscutting concerns.

Open issues For the moment, only the component level is fully integrated into the Fractal component model. When assembling the three levels, we are likely to run into consistency issues. The connection between each level is a true challenge. For example, what happens to a previously adapted component when the architecture is restructured.

The model allows various actors to add aspects at the three different levels. The architect defines a set of transformation rules. The programmer independently introduces aspects at the lower (program) level. The question of the administrator remains undefined. More likely, he will have to deal with aspects at the three levels.

The question of applying changes at design time or runtime is still open. For example, the question of adapting architecture at run-time through transformations certainly requires precautions.

6 Acknowledgment

This work was partially funded by France Telecom under the external research contract number 46 131 097.

References

1. OMG MDA specification. `http://www.omg.org/mda/specs.htm`.
2. P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection*. Elsevier Science Inc., New York, NY, USA, 1988.
3. P. Mathieu, JC. Routier, and Y. Secq. Principles for dynamic multi-agent organizations. In Kazuhiro Kuwabara and Jaeho Lee, editors, *PRIMA*, volume 2413 of *Lecture Notes in Computer Science*, Tokyo, Japan, August 2002. Springer.
4. A. Popovici, G. Alonso, and T. Gross. Spontaneous Container Services. In *ecoop*, 2003.
5. J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. , LNCS 2192*, pages 81–88. Springer-Verlag, September 2001.
6. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J-B. Stefani. An open component model and its support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering*, Edinburgh, Scotland, May 2004.
7. S. Becker and R. H. Reussner. The impact of software component adaptors on quality of service properties. In Carlos Canal, Juan Manuel Murillo, and Pascal Poizat, editors, *Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT 04)*, June 2004.
8. W. Vanderperren and D. Suvee. JAsCoAP: Adaptive programming for component-based software engineering. In Karl Lieberherr, editor, *3rd International Conference on Aspect-Oriented Software Development (AOSD-2004)*. ACM Press, March 2004.
9. R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC : An aspect-based distributed dynamic framework. *Software Practise and Experience (SPE)*, 34(12):1119–1148, October 2004.

10. N. Pessemier, L. Seinturier, and L. Duchien. Components, ADL & AOP: Towards a common approach. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution at ECOOP'04*, June 2004.
11. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., 2002.
12. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transaction on Software Engineering*, 26(1):70–93, January 2000.
13. O. Barais, L. Duchien, and A-F. Le Meur. A framework to specify incremental software architecture transformations. In *31st EUROMICRO CONFERENCE on Software Engineering and Advanced Applications (SEAA)*. IEEE Computer Society, September 2005 (to appear).
14. M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations forarchitectural connections. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 2, Washington, DC, USA, 1996. IEEE Computer Society.
15. E. Visser. A Survey of Rewritting Strategies in Program Transformation Systems. In Electronic Notes in Theoretical Computer Science, editor, *first Workshop on Reduction Strategies in Rewritting and Programming (WRS'2001)*, volume 57. Elsevier Science, May 2001.
16. ASM web site. `asm.objectweb.org`.
17. R. Douence and M. Südholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.